1 Ponteiros

" Uma base sólida de conhecimento e técnica de algoritmos é uma das características que separa o programador experiente do aprendiz. Com a moderna tecnologia de computação, é possível realizar algumas tarefas sem saber muito sobre algoritmos, mas com um boa base pode-se fazer muito, muito mais. "

- **Cormen, Leiserson, Rivest, Stein**-

Sumário:

1.1- Ponteiros

As linguagens de programação de alto nível, possuem um tipo especial de variáveis que permite manipular os endereços de memória. Essas variáveis, denominadas por **ponteiros**, são muito importantes porque dão ao programador a possibilidade de criar estruturas de dados e programas sofisticados, para aumentar e diminuar a quantidade de memória necessária para um programa ser processado em tempo de execução.

Um **ponteiro** é uma variável que armazena um endereço de memória de um objecto qualquer. Entendemos por objecto uma variável elementar, uma variável estruturada, uma função e um outro ponteiro.

Como um ponteiro é uma variável, ele precisa de ser declarado. A sua declaração é descrita pela seguinte sintaxe:

```
<tipo>  <oper> <nome>;
```

onde

<tipo> representa qualquer tipo de dados que a linguagem C suporta;
<oper> representa o operador do tipo ponteiro que pode ser ***** ou **&;**
<nome> denota uma variável;

O operador unário ***,** denominado por **operador de acesso indirecto** é um ponteiro que devolve o conteúdo do endereço de memória do seu operando. Por exemplo, nas declarações

```
int *p;
float  *f;
```

as variáveis p e f são ponteiros. O primeiro armazena o conteúdo do endereço de memória de objetos do tipo inteiro, enquanto o segundo armazena o conteúdo de memória de objectos do tipo caracter.

Um ponteiro só deve armazenar o conteúdo de um endereço de memória se esse conteúdo e o ponteiro tiverem o mesmo tipo de dados. Se essa restrição não for observada, teremos resultados imprevisíveis.

O operador unário **&,** denominado por **operador endereço** é um ponteiro que devolve o endereço de memória do seu operando. Por exemplo, no segmento de código:

```
float x = -1.0;
float  *p;
p = &x;
```

Nas duas primeiras linhas declaramos uma variável x do tipo float que recebe o valor -1.0 e um ponteiro p do mesmo tipo. Como temos uma compatibilidade de tipo de dados, na terceira linha o ponteiro p aponta para o endereço da variável x e, dizemos que p aponta para x.

Vamos consolidar esses conceitos com dois exemplos muito simples. No primeiro, pretendemos utilizar um ponteiro para mostrar o conteudo de uma variável do tipo inteiro.

```
float  x = -1.0;
float *p = &x;
*p  = 5.0;
```

Suponhamos sem perda da generalidade que durante a execução do programa, na primeira linha, seja atribuído o endereço 200 à variável x. Pela segunda linha o ponteiro p aponta para esse endereço, ou seja, p aponta para a variável que está no endereço 200. Mas na terceira linha, o valor 5.0 é atribuído a variável cujo endereço é apontado pelo ponteiro p. Como o ponteiro p aponta para o endereço 200 e como no endereço 200 temos à variável x, então a variável x recebe o valor 5.0.

No segundo exemplo, pretendemos utilizar ponteiros para calcular a soma de duas variáveis do tipo real.

```
float x = -1.0, y = 10.0,  z;
float *p = &x;
z = (* p) + y;
```

Na primeira linha, declaramos e inicializamos três variáveis do tipo float. À variável x recebe o valor -1.0, enquanto à variável y recebe o valor 10.0. Na segunda, declaramos e inicializanos um ponteiro p do tipo float. Como temos uma compatibilidade de atribuição, então o p aponta para o endereço da variável x. Na última linha, adicionamos ao conteudo do ponteiro p, que nesse momento possui o valor -1.0, o conteudo da variável y, que é igual a 10.0. Essa operação é atribuída a variável z.

1.2 - Passagem de Parâmetros

Por defeito, na linguagem C os parâmetros são passados por **valor**. Neste tipo de passagem, os parâmetros da função e os parâmetros da chamadas estão armazenados em posições de memória diferentes. Logo, nenhuma alteração aos parâmetros da função afectam os valores dos parâmetros da chamada.

Uma das principais vantagens dessa forma de passagem é que as funções ficam impedidas de aceder ao conteudos das variáveis declaradas em outras funções.

Para que uma função possa alterar os valores dos parâmetros da chamada é necessário que eles sejam passados por **referência**. Nesse caso, os parâmetros da chamada e os parâmetros da função compartilham as mesmas posições de memória.

Para clarificar esse conceito vejamos um exemplo. A função permuta() que descrevemos a seguir, troca o conteudo dos dois parâmetros passados por referência. Observe que na declaração da função os nomes dos parâmetros são precedidos por um asterístico, isso quer dizer que a função recebe como parâmetro os conteudos dos endereços representados por esses ponteiros.

```
#include <stdio.h>
...
void permuta ( int *p, int *q )
{
    int x;
    x = *p;
    *p = *q;
    *q = x;
}
```

Na chamada dessa função os parâmetros são precedidos pelo operador endereço **&,** isso quer dizer que o compilador vai transferir para a função os endereços de memória onde os parâmetros estão armazenados.

```
int main()
{
   int a = 3, b = 7;
   permuta (&a, &b );              /* Chamada da Função      */
   printf("%d %d", a, b);
   return 0;
}
```

1.3 - Aritmética de Ponteiros

À adição e à subtracção são as únicas operações aritméticas que podem ser realizadas por ponteiros. Para entender o que ocorre com a aritmética de ponteiros, vamos considerar que pt1 é um ponteiro que aponta para o endereço 2000 e que esse ponteiro é do tipo inteiro. O valor do incremento:

```
pt1++
```

depende da arquitectura do computador. Suponhamos sem perda da generalidade que o nosso computador possui uma arquitectura de 16 bits. Para essa arquitectura, o tipo caracter é armazenado em um byte, o inteiro em dois bytes, o float em quatro bytes e o double em 8 bytes. A operação anterior deverá apontar para a posição de memória do próximo elemento do tipo inteiro. O próximo elemento está no endereço 2002. O mesmo acontece com a operação de decremento, para esse caso, o ponteiro apontaria para o endereço 1998.

1.4 - Ponteiros e Vectores

Quando declaramos um vector, o compilador reserva automaticamente um bloco de bytes consecutivos de memória, para armazenar os seus elementos e, trata o nome do vector, como um ponteiro que faz referencia ao primeiro elemento. Por exemplo, na declaração

```
int x[4];
```

Se o primeiro elemento estiver armazenado no endereço 104, o segundo estará endereço 106, o terceiro no endereço 108 e o ultimo no endereço 110. Para além disso, x é um ponteiro que aponta para o endereço 104.

Como o compilador trata o nome de um vector como um ponteiro, então, acesso a qualquer elemento pode ser feito pela notação vectorial ou pela notação com de ponteiro.

Na notação vectorial, i-ésimo elemento é acedido pela variável indexada x[i], enquanto na notação por ponteiro esse elemento é acedido pelo ponteiro *(x+i).

Isto que dizer que o comando de atribuição multiplo

```
x[1] = x[2] + x[3];
```

é equivalente à:

```
*(x+1) = *(x+2) + *(x+3);
```

e que a função

```
int somaElementos ( int A[ ], int ultPos)
{
   int i, soma = 0;
   for ( i = 0; i <= ultPos; i++)  soma += A[i];
   return soma;
}
```

e equivalente à:

```
int somaElementos ( int A[ ], int ultPos)
{
   int i, soma = 0; *pta = A;
   for ( i = 0; i <= ultPos; i++)  soma += *pta;
   return soma;
}
```

A linguagem C não permite que um vector seja retornado por uma função. Para isso, é necessário declarar essa função como um ponteiro. Declaramos um ponteiro para uma função com um asterisco como prefixo do nome da função. Por exemplo:

```
char *diaSemana ( int n)
{
   static char *d[ ] = { "erro", "domingo", "segunda-feira", "terça-feira",
                         "quarta-feira", "quinta-feira", "sexta-feira", "sábado" };
    return d [ 1<=n && n<=7 ? n : 0 ];
}
```

Quando a função receber como parâmetro um número inteiro entre 1 a 7, ela retornará o correspondente dia da semana por extenso. Mas quando a função receber valores fora desse intervalo, retornará a cadeira de caracteres "erro".

1.5 - Ponteiros e String's

As cadeias de caracteres são vectores. Como a notação por ponteiros é mais eficiente do que a notação vectorial para aceder aos elementos de um vector, devemos utilizá-la para desenvolver as nossas funções. Por exemplo, para a função "string compare" temos:

```
void strcpy(char *s, char *t)
{
  while ((*s = *t) != ´\0´)
  {
     s++;
      t++;
  }
}
```

que pode ser optimizada para:

```
void strcpy(char *s, char *t)
{
   while ((*s++ = *t++));
}
```

Os vectores de ponteiros são normalmente utilizados para processar vectores de cadeiras de caracteres. Um dos exemplos muito comuns em programação, é a impressão das mensagens de erro de uma aplicação. Em vez de escrevermos comandos para imprimir essas mensagens no interior das funções que desenvolvemos, devemos retornar um código de erro, e tratá-lo num procedimento independente.

Suponhamos sem perda da generalidade, que pretendemos desenvolver um programa para movimentar um rato num labirinto e necessitamos de mostrar as

seguintes mensagens: Posição inicial inválida, Posição final não encontrada e Posição final encontrada. O procedimento que descrevemos a seguir implementa o tratamento dos erros enviados pelas funções que compõem esse programa.

```
void msgErro ( int codigo )
{
   static char * Erro[ ] = { "Posicao Inicial Inválida",
                             "Posicao Final Nao Encontrada",
                             "Posicao Final Encontrada" };
   printf ( "\n "%s", Erro[codigo] );
}
```

Observe que o vector Erro contêm um ponteiro para cada string. Se o procedimento msgErro() receber como parâmetro de entrada o número dois, ele irá mostrar na tela a mensagem Posição Final Encontrada.

1.6 - Ponteiros e Funções

Em todas as linguagens de programação de alto nível, podemos criar ponteiros para variáveis que armazenam endereços de memória onde está localizada linhas de código.

Mas a linguagem C vai mais além, ela permite que se crie ponteiros que apontam para funções, ou seja, ponteiros que armazenam endereços de memória que indicam o início do código de uma função.

Vimos que quando passamos um vector como parâmetro de uma função, o compilador trata o nome desse vector como um ponteiro para o primeiro endereço de memória.

Para as funções, o compilador trata essa passagem da mesma maneira, ou seja, ele associa um ponteiro ao endereço de memória onde inicia o código da função. Por exemplo, a função

```
int soma(int a, int b)
{
   return (a+b);
}
```

recebe como parâmetro duas variáveis do tipo inteiro e devolve como retorno da função a soma do conteúdo essas variáveis. Na declaração

```
int (*pts)(int, int );
```

criamos ponteiro pts para uma função que recebe como parâmetro duas variáveis do tipo inteiro e devolve como retorno da função um valor do mesmo tipo. Se executássemos o segmento de código:

```
pts = soma;
printf ("%d, %d\n",pts(4,5));
```

associamos o ponteiro pts a função soma e na linha seguinte, mostravamos na tela o valor 9. Observe que na chamada pts(4,5) estamos a invovar a função soma() com os argumentos 4 e 5.

Para consolidar esse conceito, vamos descrever mais um exemplo:

```
#include <stdio.h>
#include <math.h>
int main ()
{
   double (*p)(double);
   p = sqrt;
   printf ("%.1f", p(25));
   return 0;
}
```

Na declaração double (*p)(double) declaramos um ponteiro p para apontar para uma função do tipo double, que recebe um parâmetro um tipo double. Em seguida, o endereço da função sqrt() é atribuído a p. Quando efectuarmos a chamada p(25) estamos na verdade a chamar a função sqrt(25). Como essa chamada esta a ser executada no interior da função printf() ela mostrará na tela o valor 5.

Um dos recursos muito valiosos da linguagem C é a passagem do nome de uma função como parâmetro para outras funções. Este recurso dá-nos a possibilidade de trocar a função que entra como parâmetro na chamada da outra função. Por exemplo na declaração

```
int f ( double g( int x), int a);
```

a função f possui dois parâmetros de entrada. O primeiro é uma função que recebe como parâmetro de entrada um número inteiro e devolve como retorno da função um número real, enquanto o seguindo é uma variável do tipo inteiro.

Lembre-se mais uma vês, que o compilador trata os nomes das funções como ponteiros. Então, essa declaração é equivalente a

```
int f ( double (*g)( int x), int a);
```

onde g é um ponteiro para uma função que recebe como parâmetro um inteiro e devolve como valor de retorno um número real.

Vamos consolidar esse conceito com um exemplo no domínio da análise numérica. O método de Newton é um algoritmo muito eficiente para determinar as raízes (zeros) de funções contínuas. Contudo antes de implementa-lo enunciaremos dois teoremas da análise matemática.

Teorema 1: Seja f é uma função contínua. Dizemos que um número x é uma raiz de f se e somente se f(x) = 0.

Teorema 2: Dado o intervalo fechado [a..b], se f(a) > 0 e f(b) < 0, então existe pelo mesmo um ponto interior c nesse intervalo tais que f(c) = 0;

Agora, faremos uma breve descrição sobre o método. O método de Newton determina um desses pontos e possui o seguinte funcionamento. Vamos assumir que f(a) > 0 e que f(b) < 0. Calculamos um ponto intermédio m tais que m = (a+b)/2. Se f(m) = 0, o ponto m é a raiz desejada. Se f(m) > 0 então a raiz deve estar entre os pontos m e b, visto que f(m) > 0 e se f(b) < 0. No caso contrário, a raiz deve está entre a e m. Para qualquer hipótese, o intervalo onde a raiz se encontra foi dividido ao meio. Este processo continua até que a raiz seja encontrada.

Essa descrição permite desenvolver a seguinte função.

```
double Newton (double f(double), double a, double b)
{
  double m, x;
  do {
        m = (a+b)/2.0;
        x = f(m);
        if ( (fabs(x < EPSILON )) || (fabs (b - a < EPSILON) ) ) break;
        if (f(a)*x > 0.0)
           a=m;
        else
            b=m;
     }
  while (1);
  return m;
}
```

onde

```
#define EPSILON 1.0E-10                    /* erro de aproximação */
```

Essa função deve determinar correctamente uma das raízes para qualquer função f que seja passada como parâmetro, desde que essa função tenha nos extremos do intervalo [a,b] valores com sinais trocados.

Como a função Newton() tem por finalidade determinar uma única raiz , é da responsabilidade do programador garantir que essa função só deve ser invocada quando ela tiver valores opostos nos extremos do intervalo.

Se quiséssemos calcular o valor do cosseno de 0 a pi, em radianos, basta desenvolver uma função principal que entre outras instruções teria o seguinte segmento de código:

```
printf ("Funcao cosseno entre 0 e pi:\n");
raiz = Newton (cos,0.0,3.141592);
printf (" \t Raiz no ponto %g\n ", raiz);
printf (" \t Valor de cos(%g) = %g\n ", raiz, cos(raiz));
```

Observe nos parâmetros de chamada da função Newton. O primeiro é o nome da função cosseno. Ao invocar a função Newton, o nome do parâmetro f é substituído pela função cosseno e a função newton calcula a raiz que pretendemos. Contudo, para realizar esse cálculo devemos incluir a biblioteca math.h.

Mas se quiséssemos utilizar a função Newton para calcular a raiz de uma função polinomial de um grau qualquer, basta desenvolver uma função principal que entre outras instruções teria o seguinte segmento de código

```
printf ( "Funcao polinomial entre 0 e 10.0:\n" );
raiz = Newton (poli,0.0,10.0);
printf ( "\t Raiz no ponto %g\n ",raiz);
printf ( "\t Valor de poli(%g) = %g\n ", raiz, poli(raiz));
```

onde poli() é uma função que recebe como argumento um número real e devolve como retorno da função o valor do polinómio nesse número.

Vamos estudar mais um exemplo. Suponhamos que pretendemos desenvolver um programa para calcular o valor máximo e o valor mínimo de um vector com números inteiros. Uma das estratégias vistas num primeiro curso, consiste em desenvolver uma função para determinar o valor mínimo

```
int  min (int v[ ], int ultPos)
{
   int i, aux =v[0];
   for (i =1; i<ultPos; i++)
   if ( aux > v[i] )  aux = v[i];
   return aux;
}
```

e uma outra função para determinar o valor máximo.

```
int max (int v[ ], int ultPos)
{
   int i, aux = v[0];
   for (i = 1; i < ultPos; i++)
   if ( aux < v[i] )  aux = v[i];
   return aux;
}
```

Mas, essas funções têm practicamente as mesmas linhas de código. As únicas diferenças estão nos seus nomes e na expressão do comando condicional.
Para evitar essa redundância de código devemos utilizar o **polimorfismo** que é uma das bases da programação orientada por objectos.

```
#include <stdio.h>
...

int menor (int x, int y)
{
   return x > y;
}

int maior (int x, int y)
{
   return x < y;
 }

int minimax (int v[ ], int ultPos, int (*cmp)(int, int))
{
   int i, x =v[0];
   for( i =1; i < ultPos; i++)
   if( cmp(x,v[i]) )  x = v[i];
   return x;
}
```

Agora, vamos desenvolver uma função principal para determinar o valor máximo e o valor mínimo de um vector.

```
int main()
{
   int w[5] = {78,34,12,45,51};
   printf ( "%d ", minimax(w,5,menor) );
   printf ("%d ", minimax(w,5,maior) );
   return 0;
}
```

Com essa técnica, criamos uma terceira função que denominamos por minimax() cujo terceiro parâmetro é um ponteiro para uma função do tipo inteiro que recebe dois argumentos do mesmo tipo. Esse parâmetro permite encontrar o valor máximo ou o valor mínimo do vector.

1.7 - Ponteiros e Estruturas

Um ponteiro para uma estrutura não é diferente de um ponteiro para um tipo de dados elementar. Se p aponta para uma estrutura que tem um campo m, então podemos escrever (∗p).m para aceder a esse campo. Por exemplo:

```
#include <stdio.h>
...
```

```
typedef struct
{
   int ordenada;
   int absissa;
 } TPlano;


int main()
{
    TPlano *p, quadrante = { 1, 3 };
    p = &quadrante;
    printf("{ %s, %d }", (*p).ordenada, (*p).absissa );
}
```

Como o operador ponto (.) tem maior precedência que o operador de acesso indirecto (∗), temos de utilizar obrigatóriamente parênteses para fazer referencia a qualquer campo dessa estrutura no comando printf().

Para evitar essa notação pesada, a linguagem C possui um operador específico para aceder aos campos das estruturas, denominado por operador seta. Com esse operador, em vez de escrevermos (∗ponteiro).membro passamos a escrever ponteiro−>membro. Então, o programa anterior deveria ser escrito como:

```
int main()
{
   TPlano *p, quadrante = { 1, 3 };
    p = &quadrante;
    printf("{ %s, %d }", p->ordenada, p->absissa );
}
```

Vamos estudar em seguida, algumas operações com estruturas. As funções podem receber e retornar via valor de retorno estruturas.

```
TPlano adicionarPontos (TPlano p1, TPlano p2)
{
   TPlano res;
    res.x = p1.x + p2.x;
    res.y = p1.y + p2.y;
    return res;
}
```

Observe no entanto que a função retorna uma cópia da estrutura, que neste caso é a estrutura res. Como a passagem é feita por valor o acesso a qualquer campo é feito pelo operador ponto.

Mas as estruturas passadas por valor podem trazer alguns problemas de eficiência. De facto a passagem de estruturas muito grandes como parâmetros é ineficiente, uma vez que é necessário efectuar cópia de todos os campos. Por essa razão, utilizam-se normalmente ponteiros para estruturas e, portanto,

a passagem passa a ser por referência. Nesse sentido, o exemplo anterior teria a seguinte código:

```
void adicionarPonto (TPlano *p1, TPlano *p2, Tplano *res)
{
  res->x = p1->x + p2->x;
  res->y = p1->y + p2->y;
}
```

1.7 - Exercícios

1.7.1- Desenvolva uma função que recebe como parâmetro um vector v de números reais e a posição do último elemento inserido. Devolve como valor de retorno o elemento máximo e o elemento mínimo.

1.7.2- Desenvolva uma função que recebe como parâmetro um vector de números inteiros e um inteiro n maior ou igual a zeros. Carregar nesse vector as n notas das provas dos estudantes.

1.7.3- Para além da adição de ponteiros, uma outra operação possível é a subtração entre ponteiros do mesmo tipo. Quando um ponteiro é subtraído de um outro, o resultado é o número de elementos existentes no espaço de memória compreendido entre os endereços apontados por eles. Utilizando essa ideia, desenvolva a função strlen(s), que devolve o número de

1.7.4 - Desenvolva a função strchr(s,c) que devolve o endereço da primeira ocorrência do caracter c na string s ou o valor NULL caso esse caracter não seja encontrado.

1.7.5 - Desenvolva a função strsub(s,i,n) que devolve uma substring de s que inicia na posição i e tem no máximo, n caracteres. A string s não deve ser alterada.

1.7.6 - Defina uma estrutura para representar um livros, que entre outros campos deve conter o título, o autor e oano de publicação. Em seguida, desenvolva em seguida, uma função para carregar essa estrutura com dados digitados pela teclado.

1.7.7- Dados dois polinómios p(x) e g(x) de grau m e n, com coeficientes inteiros, armazenados nos vectores p[m] e g[n]. Desenvolva as funções para realizar as operações de adição, subtração, multiplicação e divisão.

1.7.8- Defina uma estrutura para representar um número complexo. Em seguida, desenvolva as funções para realizar as operações de adição, subtração, divisão e subtração.